Replication in DHTs using Dynamic Groups

Reza Akbarinia, Mounir Tlili, Esther Pacitti, Patrick Valduriez, Alexandre A. B. Lima

Replication in DHTs using Dynamic Groups

Reza Akbarinia[1], Mounir Tlili[2], Esther Pacitti[1], Patrick Valduriez[1], Alexandre A. B. Lima[3]

[1]INRIA and LIRMM, Montpellier, France
[2]INRIA and LINA, Univ. Nantes, France
[3]COPPE/UFRJ, Rio de Janeiro, Brazil
Reza.Akbarinia@inria.fr, Mounir.Tlili@univ-nantes.fr, pacitti@lirmm.fr, Patrick.Valduriez@inria.fr, assis@cos.ufrj.br

Abstract. Distributed Hash Tables (DHTs) provide an efficient solution for data location and lookup in large-scale P2P systems. However, it is up to the applications to deal with the availability of the data they store in the DHT, e.g. via replication. To improve data availability, most DHT applications rely on data replication. However, efficient replication management is quite challenging, in particular because of concurrent and missed updates. In this paper, we propose a complete solution to data replication in DHTs. We propose a new service, called Continuous Timestamp based Replication Management (CTRM), which deals with the efficient storage, retrieval and updating of replicas in DHTs. In CTRM, the replicas are maintained by groups of peers which are determined dynamically using a hash function. To perform updates on replicas, we propose a new protocol that stamps the updates with timestamps that are generated in a distributed fashion using the dynamic groups. Timestamps are not only monotonically increasing but also continuous, i.e. without gap. The property of monotonically increasing allows applications to determine a total order on updates. The other property, i.e. continuity, enables applications to deal with missed updates. We evaluated the performance of our solution through simulation and experimentation. The results show its effectiveness for replication management in DHTs.

1 Introduction

Distributed Hash Tables (DHTs) such as CAN [17], Chord [21] and Pastry [20], provide an efficient solution for data location and lookup in large-scale P2P systems. While there are significant implementation differences between DHTs, they all map a given key *k* onto a peer *p* using a hash function and can lookup *p* efficiently, usually in O(*log n)* routing hops, where *n* is the number of peers [5]. One of the main characteristics of DHTs (and other P2P systems) is the dynamic behavior of peers which can join and leave the system frequently, at any time. When a peer gets offline, its data becomes unavailable. To improve data availability, most applications which are built on top of DHTs rely on data replication by storing the (*key, data*) pairs at several peers, *e.g.* using several hash functions. If one peer is unavailable, its data can still be retrieved from the other peers that hold a replica. However, update management is difficult because of the dynamic behaviour of peers and concurrent updates. There may be *replica holders* (i.e. peers that maintain replicas) that do not receive the updates, e.g. because they are absent during the update operation. Thus, we need a mechanism that efficiently determines whether a replica on a peer is up-to-date, despite missed updates. In addition, to deal with concurrent updates, we need to determine a total order on the update operations.

* Work partially funded by the ANR DataRing projet.

In this paper, we give an efficient solution to replication management in DHTs. We propose a new service, called Continuous Timestamp based Replication Management (CTRM), which deals with the efficient storage, retrieval and updating of replicas in DHTs. In CTRM, the replicas are maintained by groups of peers, called replica holder groups, which are dynamically determined using a hash function. To perform updates on replicas, we propose a new protocol that stamps the updates with timestamps that are generated in a distributed fashion using the members of the groups. The updates' timestamps are not only monotonically increasing but also continuous, i.e. without gap. The property of monotonically increasing allows CTRM to determine a total order on updates and to deal with concurrent updates. The continuity of timestamps enables replica holders to detect the existence of missed updates by looking at the timestamps of the updates they have received. Examples of applications that can take advantage of continuous timestamping are the P2P collaborative text editing applications, e.g. P2P Wiki [23], which need to reconcile the updates done by collaborating users. We analyze the network cost of CTRM using a probabilistic approach, and show that its cost is very low in comparison to two baseline services in DHTs. We evaluated CTRM through experimentation and simulation; the results show its effectiveness. In our experiments, we compared CTRM with two baseline services, and the results show that with a low overhead in update response time, CTRM supports fault-tolerant data replication using continuous timestamps. The results also show that data retrieval with CTRM is much more efficient than the baseline services. We investigated the effect of peer failures on the correctness of CTRM and the results show that it works correctly even in the presence of peer failures.

This paper is an extended version of [2] that involves at least 38% of new material including the following contributions. First, in Section 4, we extend the concept of replica holder groups which are essential for our solution. In particular, we deal with the dynamic behaviour of the group members, which can leave the system at any time. Second, in Section 6, we give a communication cost analysis of our solution, using a probabilistic approach, and compare the cost of our solution with those of two baseline services. We also include more discussion in Section 8 about related work on replication management in P2P systems.

The rest of this paper is organized as follows. In Section 2, we define the problem we address in this paper. In Section 3, we give an overview of our CTRM service and its operations. In Section 4, we describe the replica holder groups in CTRM. In Section 5, we propose the new UCT protocol which is designed for updating replicas in CTRM. Section 6 presents a cost analysis of the CTRM service. Section 7 reports a performance evaluation of our solution. Section 8 discusses related work, and Section 9 concludes.

2 Problem Definition

In this paper we deal with improving data availability in DHTs. Like several other protocols and applications designed over DHTs, e.g. [5], we assume that the lookup service of the DHT behaves properly. That is, given a key k, it either finds correctly the responsible for k or reports an error, *e.g.* in the case of network partitioning where the responsible peer is not reachable.

To improve data availability, we replicate each data at a group of peers of the DHT which we call *replica holders*. Each replica holder keeps a *replica copy* of a replicated data. Each replica may be updated locally by a replica holder or remotely by other peers of the DHT. This model is in conformance with the *multi-master replication* model [15].

The problem that arises is that a replica holder may fail or leave the system at any time. Thus, the replica holder may miss some updates during its absence. Furthermore, updates on different replicas of a data may be performed in parallel, i.e. concurrently. To ensure consistency, updates must be applied to all replicas in a specific total order.

In this model, to ensure eventual consistency of replicas, we need a distributed mechanism that determines 1) a total order for the updates; 2) the number of missed updates at a replica holder. Such a mechanism allows dealing with concurrent updates, i.e. committing them in the same order at all replica holders. In addition, it allows a rejoining (recovering) replica holder to determine whether its local replica is up-to-date or not, and how many updates should be applied on the replica if it is not up-to-date.

In this paper, we aim at developing a replication management service supporting the above-mentioned mechanism in DHTs. One solution for realizing such a mechanism is to stamp the updates with timestamps that are monotonically increasing and continuous. We call such a mechanism *update with continuous timestamps*.

Let *patch* be the action (or set of actions) generated by a peer during one update operation. Then, the property of update with continuous timestamps can be defined as follows.

Definition 1: Update with continuous timestamps (UCT). An update mechanism is UCT *iff* : the update patches are stamped by increasing real numbers such that the difference between the timestamps of any two consecutive committed updates is one.

Formally, consider two consecutive committed updates u_1 and u_2 on a data d, and let pch_1 and pch_2 be the patches of u_1 and u_2, respectively. Assume that u_2 is done after u_1, and let t_1 and t_2 be the timestamps of pch_1 and pch_2 respectively. Then we should have $t_2 = t_1 + 1$;

To support the UCT property in a DHT, we must deal with two challenges: 1) To generate continuous timestamps in the DHT in a distributed fashion; 2) To ensure that any two consecutive generated timestamps are used for two consecutive updates. Dealing with the first challenge is hard, in particular due to the dynamic behavior of peers, which can leave or join the system at any time and frequently. This behavior makes inappropriate the timestamping solutions based on physical clocks, because the distributed clock synchronization algorithms do not guarantee good synchronization precision if the nodes are not linked together long enough [16]. Addressing the second challenge is difficult as well, because there may be generated timestamps which are used for no update, e.g. because the timestamp requester peer may fail before doing the update.

3 Overview of Replication Management in CTRM

CTRM (Continuous Timestamp based Replication Management) is a replication management service which we designed to deal with efficient storage, retrieval and updating of replicas on top of DHTs, while supporting the UCT property.

To provide high data availability, CTRM replicates each data in the DHT at a group of peers, called *replica holder group*. For each replicated data, there is a replica holder group which is determined dynamically by using a hash function. To know the group which holds the replica of a data, peers of the DHT apply the hash function on the data ID, and using the DHTs lookup service to find the group. The details of the replica holder groups are presented in Section 4.

3.1 Data Update

CTRM supports multi-master data replication, i.e. any peer in the DHT can update the replicated data. After each update on a data by a peer p, the corresponding patch, i.e. set of update actions, is sent by p to the replica holder group where a monotonically increasing timestamp is generated by one of the members, i.e. the responsible for the group. Then the patch and its timestamp are published to the members of the group using an update protocol, called UCT protocol. The details of the UCT protocol are presented in Section 5.

3.2 Replica Retrieval

To retrieve an up-to-date replica of a data, the request is sent to the peer that is responsible for the data's replica holder group. The responsible peer sends the data and the latest generated timestamp to the group members, one by one. The first member that maintains an up-to-date replica returns it to the requester. To check whether their replicas are up-to-date, replica holders check the two following conditions, called *up-to-date conditions*:

1. The latest generated timestamp is equal to the timestamp of the latest patch received by the replica holder.
2. The timestamps of the received patches are continuous, i.e. there is no missed update.

The UCT protocol, which is used for updating the data in CTRM, guarantees that if at peer p there is no gap between the timestamps and the last timestamp is equal to the last generated one, then p has received all replica updates. In contrast, if there is some gap in the received timestamps, then there should be some missed updates at p.

If during the replica retrieval operation, a replica holder p understands that it misses some updates, then it retrieves the missed updates and their timestamps from the group's responsible peer or other members that hold them, and updates its replica.

In addition to the replica retrieval operation, the up-to-date conditions are also verified periodically by each member of the group. If the conditions do not hold, the member updates its replica by retrieving the missed updates from other members of the group.

4 Replica Holder Groups

Replica holder groups are dynamic groups of peers which are responsible for maintaining the replicas of data, timestamping the updates, and returning up-to-date data to the users.

In this section, we first describe the idea behind the replica holder groups, then discuss on how they assure their correct functionality in the presence of peer join/departures, which can be frequent in P2P systems.

4.1 Basic Ideas

Let G_k be the group of peers that maintain the replicas of a data whose ID is k. We call these peers the *replica holder group of k*. For each group, there is a responsible peer which is also one of its members. For choosing a responsible peer for the group

G_k, we use a hash function h_r, and the peer p that is responsible for key=$h_r(k)$ in the DHT, is the responsible for G_k. In this paper, the peer that is responsible for key=$h_r(k)$ is denoted by *rsp(k , h_r), i.e.* called responsible for k with regard to hash function h_r. In addition to *rsp(k , h_r)*, some of the peers that are close to it, .e.g. its neighbors, are members of G_k. Each member of the group knows the address of other members of the group. The number of members of a replica holders group, i.e. $|G_k|$, is a system's parameter.

Each group member p periodically sends alive messages to the group's responsible peer, and the responsible peer returns to it the current list of members. If the responsible peer does not receive an alive message from a member, it assumes that the member has failed. When a member of a group leaves the system or fails, after getting aware of this departure, the responsible peer invites a close peer to join the group, *e.g.* one of its neighbors. The new member receives from the responsible peer a list of other members as well as up-to-date replicas of all data replicated by the group.

The peer p that is responsible for G_k generates timestamps for the updates done on the data k. For generating the timestamps, it uses a local counter called *counter of k at p* which we denote as $c_{p,k}$. When p receives an update request for a data k, it increments the value of $c_{p,k}$ by one and stores the update patch and the timestamp over the other members of the group using a protocol which we describe in Section 5.

In the situations where the group's responsible peer leaves the system or fails, another peer takes it over. This responsibility change can also happen in the situations where another peer joins the system and becomes responsible for the key $h_r(k)$ in the DHT. In the next section, we discuss on how the responsibility migrates in these situations, and how the new responsible peer initializes its counter to the correct timestamp value, i.e. to the value of the last generated timestamp.

4.2 Dealing with Departure of Group's Responsible Peer

The responsible peer is the most important member of the group. In the management of the replica groups, we must deal with the cases where the responsible peer leaves the system or fails. The main issues are: how to determine the next responsible peer, and how to initialize the counter values on it.

4.2.1 Who Is the Next Group's Responsible?

As discussed in Section 4.1, the responsible for the group G_k is the peer that is responsible for the key $h_r(k)$ in the DHT. Notice that at any time, there is a responsible peer for each key. If the current responsible for the key $h_r(k)$ leaves the DHT, another peer, say p, becomes responsible for the key. This peer p becomes also the new responsible for the group G_k. Therefore, if a peer wants to contact the responsible for G_k, the lookup service of the DHT gives it the address of p.

An interesting question is about the relationship between the current and the next responsible peer in the DHT. To answer the question, we observe that, in DHTs, the next peer that obtains the responsibility for k is typically a neighbor of the current responsible peer, so the next responsible peer is one of the members of the group. We now illustrate this observation with CAN and Chord, two popular DHTs.

Let *rsp(k, h_r)* be the current responsible peer for group G_k , and *nrsp(k, h_r)* be the one that takes it over. Let us assume that peer q is *rsp(k, h_r)* and peer p is *nrsp(k, h_r)*. In CAN and Chord, there are only two ways by which p would obtain the responsibility for k. First, q leaves the P2P system or fails, so the responsibility of k is assigned to p. Second, p joins the P2P system which assigns it the responsibility for k, so q looses the responsibility for k despite its presence in the P2P system. In both

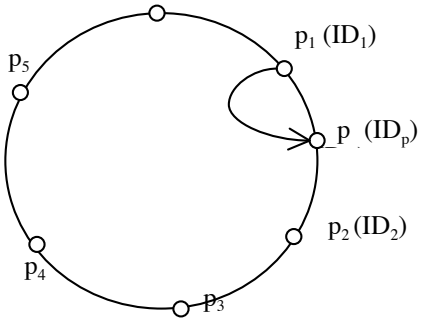

Figure 1. Responsibility migration in Chord

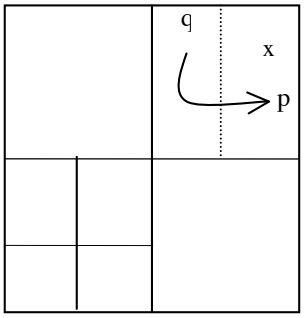

Figure 2. Responsibility migration in CAN, based on a two dimensional coordinate space

cases, we show that both CAN and Chord have the property that *nrsp(k, h_r)* is one of the neighbors *rsp(k, h_r)*.

Chord. In Chord [21], each peer has an *m*-bit identifier (ID). The peer IDs are ordered in a circle and the neighbors of a peer are the peers whose distance from *p* clockwise in the circle is 2^i for $0 \leq i \leq m$. The responsible for $h_r(k)$ is the first peer whose ID is equal or follows $h_r(k)$. Consider a new joining peer *p* with identifier ID_p. Suppose that the position of *p* in the circle is just between two peers p_1 and p_2 with identifiers ID_1 and ID_2, respectively. Without loss of generality, we assume that $ID_1 < ID_2$, thus we have $ID_1 < ID_p < ID_2$. Before the entrance of *p*, the peer p_2 was responsible for *k* if and only if $ID_1 < h_r(k) \leq ID_2$. When *p* joins Chord, it becomes responsible for *k* if and only if $ID_1 < h_r(k) \leq ID_p$ (see Figure 1). In other words, *p* becomes responsible for a part of the keys for which p_2 was responsible. Since the distance clockwise from *p* to p_2 is 2^0, p_2 is a neighbor of *p*. Thus, in the case of join, the next responsible peer is one of the neighbors of the current responsible. If *p* leaves the system or fails, the next peer in the circle, say p_2, becomes responsible for its keys.

CAN. We show this property by giving a brief description of CAN's protocol for joining and leaving the system [17]. CAN maintains a virtual coordinate space partitioned among the peers. The partition which a peer owns is called its zone. According to CAN, a peer *p* is responsible for $h_r(k)$ if and only if $h_r(k)$ is in *p*'s zone. When a new peer, say *p*, wants to join CAN, it chooses a point *X* and sends a join request to the peer whose zone involves *X*. The current owner of the zone, say *q*, splits its zone in half and the new peer occupies one half, then *q* becomes one of *p*'s neighbors (see Figure 2). Thus, in the case of join, *nrsp(k, h_r)* is one of the neighbors of *rsp(k, h_r)*. Also, when a peer *p* leaves the system or fails, its zone will be occupied by one of its neighbors, *i.e.* the one that has the smallest zone. Thus, in the case of leave or fail, *nrsp(k, h_r)* is one of the neighbors of *rsp(k, h_r)*, and that neighbor is known for *rsp(k, h_r)*.

Following the above discussion, in Chord or CAN when the current group's responsible peer leaves the system or fails, one of its neighbors becomes the next responsible peer.

4.2.2 Timestamp Initialization

In the case where a responsible peer q leaves the system or fails, the next responsible p should initialize its counters to the last value of generated timestamps. In CTRM, we consider two different situations for counter initialization: 1) normal departure of q; 2) failure of q.

Normal Departure

When a responsible peer leaves the system normally, i.e. without failure, the counter initialization is done by directly transferring the counters from the current responsible peer to the next one at the end of its responsibility.

Let q and p be two peers, and $K' \subseteq K$ be the set of keys for which q is the current responsible peer, and p is the next responsible. Once q reaches the end of its responsibility for the keys in K', *e.g.* before leaving the system, it sends to p all its counters that have been initialized for the keys involved in K'.

Failure

In the cases where a responsible peer fails, the next responsible peer uses the timestamp values, which are stored along with updates over the members of the group, in order to initialize its counters. Let k be a key whose responsible fails, and p be the peer that is the new responsible for it. For initializing the counter of k, the new responsible peer p contacts the members of the group, retrieves the most recent timestamp which is stored over each member, and selects the highest timestamp as the value of the counter for k, i.e. $c_{p,k}$.

One important question is how the new responsible peer p gets the address of the other members? The answer is as follows. If p was a member of the group before becoming its responsible, it has the address of other members of the group. Thus, it can communicate with them easily. If it is a new member, *i.e.* it has just joined the DHT, it waits until being contacted by the other members of the group. Recall that each member of the group, e.g. q, periodically sends an alive message to the group's responsible peer. If q receives no acknowledge from the responsible peer, it understands that probably the responsible peer has failed. Thus, it uses the lookup service of the DHT to find the address of the peer that is the responsible for $h_r(k)$ in the DHT. If the address is different from the previous one, q gets sure that the responsible peer has changed, so it sends it a message that involves the address of other members of the group. It also contacts the other members of the group to inform them about the modification in the responsibility of the group.

5 Update with Continuous Timestamps

To update the replicated data in the replica holder groups, CTRM uses a new protocol called UCT (Update with Continuous Timestamps). In this section, we describe the details of UCT.

5.1 UCT Protocol

To simplify the description of our UCT protocol, we assume the existence of (not perfect) failure detectors [7] that can be implemented as follows. When we setup a failure detector on a peer p to monitor peer q, the failure detector periodically sends

1. On update requester:
   - Send *{k, pch}* to *rsp(k, h_r)*
   - Monitor *rsp(k, h_r)* using a failure detector
   - Go to Step 8 if *rsp(k, h_r)* fails

2. On *rsp(k, h_r)*: upon receiving *{k, pch}*
   - Set $c_k = c_k + 1$; // increase counter by one
     // initially we have c_k=0;
   - Let ts = c_k, send *{k, pch, ts}* to other replica holders;
   - Set a timer on, called ackTimer, to a default time

3. On each replica holder: upon receiving *{k, pch, ts}*
   - Maintain *{k, pch, ts}* in a temporary memory on disk;
   - Send ack to *rsp(k, h_r)*;

4. On *rsp(k, h_r)*: upon expiring ackTimer
   - If (number of received acks ≥ threshold δ) then send "commit" message to the replica holders;
   - Else set $c_k = c_k - 1$, and send "abort" message to the update requester;

5. On each replica holder: upon receiving "commit"
   - Maintain *{pch, ts}* as a committed patch for *k*.
   - Update the local replica using *pch*;
   - Send "terminate" message to *rsp(k, h_r)*

6. On *rsp(k, h_r)*: upon receiving the first 'terminate' message
   - Send "terminate" to update requester

7. On update requester: receiving the 'terminate' from *rsp(k, h_r)*
   - Commit the update operation

8. On update requester: upon detecting a failure on *rsp(k, h_r)*
   - If the 'terminate' message is received then commit the update operation;
   - Else, check replica holders, if at least one of them received the 'commit' message then commit the update operation;
   - Else, abort the update operation;

Figure 3. UCT protocol

ping messages to *q* in order to test whether *q* is still alive (and connected). If the failure detector receives no response from *q*, then it considers *q* as a failed peer, and triggers an error message to inform *p* about this failure.

Let us now describe the UCT protocol. Let p_0 be the peer that wants to update a data whose ID is *k*. The peer p_0 is called update requester. Let *pch* be the patch of the update performed by p_0. Let p_1 be the responsible for the replica holder group of *k*, *i.e.* p_1= *rsp(k, h_r)*. The protocol proceeds as follows (see Figure 3):

- **Update request.** In this phase, the update requester, *i.e.* p_0, obtains the address of the group's responsible peer, *i.e.* p_1, by using the DHT's lookup service, and sends to it an update request containing the pair *(k, pch)*. Then, p_0 waits for a commit message from p_1. It also uses a failure detector and monitors p_1. The wait time is limited by a default value, *e.g.* by using a timer. If p_0 receives the terminate message from p_1, then it commits the operation. If the timer expires or the failure detector reports a fault of p_1, then p_0 checks whether the update has been done or not, *i.e.* by checking the data at replica holders. If the answer is positive, then the operation is committed, else it is aborted.

- **Timestamp generation and replica publication.** After receiving the update request, p_1 generates a timestamp for *k*, *e.g.* *ts*, by increasing a local counter that it keeps for *k*, say c_k. Then, it sends *(k, pch, ts)* to the replica holders, *i.e.* the members of its group, and asks them to return an acknowledgement. When a replica holder receives *(k, pch, ts)*, it returns the acknowledgement to p_1 and maintains the data in a temporary memory on disk. The patch is not considered as an update before receiving a commit message from p_1. If the number of received acknowledgements is more than or equal to a threshold δ, then p_1 starts the update confirmation phase. Otherwise p_1 sends an abort message to p_0. The

threshold δ is a system parameter, e.g. it is chosen in such a way that the probability that δ peers of the group simultaneously fail is almost zero.

- **Update confirmation.** In this phase, p_1 sends the commit message to the replica holders. When a replica holder receives the commit message, it labels *{pch, ts}* as a committed patch for *k*. Then, it executes the patch on its local replica, and sends a terminate message to p_1. After receiving the first terminate message from replica holders, p_1 sends a terminate message to p_0. If a replica holder does not receive the commit message for a patch, it discards the patch upon receiving a new patch containing the same or greater timestamp value.

Notice that the goal of our protocol is not to provide eager replication, but to have at least δ replica holders that receive the patch and its timestamp. If this goal is attained, the update operation is committed. Otherwise it is aborted, and the update requester should try its update later.

Let us now consider the case of concurrent updates, e.g. two or more peers want to update a data *d* at the same time. In this case, the concurrent peers send their request to the peer that is responsible for the *d*'s group, say p_1. The peer p_1 determines an order for the requests, e.g. depending on their arrival time or on the distance of requesters if the requests arrive at the same time. Then it processes the requests one by one according their order, i.e. it commits or aborts one request and starts the next one. Thus, concurrent updates make no problem of inconsistency for our replication management service.

5.2 Fault Tolerance of UCT protocol

Let us now study the effect of peer failures on the UCT protocol and discuss how they are handled. By peer failures, we mean the situations where a peer crashes or gets disconnected from the network abnormally, e.g. without informing the responsible peer. We show that these failures do not block our update protocol. We also show that even in the presence of these failures, the protocol guarantees continuous timestamping, *i.e.* when an update is committed, the timestamp of its patch is only one unit greater than that of the previous one. For this, it is sufficient to show that if the group's responsible peer fails, each generated timestamp is attached with a committed patch, or is aborted. By aborting a timestamp, we mean returning the counter's value to its value before the update operation. During the UCT protocol execution, a failure on the group's responsible peer may happen in one of the following time intervals:

- **I_1: after receiving the update request and before generating the timestamp.** If the group's responsible peer fails in this interval, then after some time, the failure detector detects the failure or the timer timeouts. Afterwards, the update requester checks the update at replica holders, and since it has not been done, the operation is aborted. Therefore, a failure in this interval does not block the protocol, and continuous timestamping is assured because no update is performed.

- **I_2: after I_1 and before sending the patch to replica holders.** In this interval, like in the previous one, the failure detector detects the failure or the timer timeouts, and thus the operation is aborted. The timestamp *ts*, which is generated by the failed responsible peer, is aborted as follows. When the responsible peer

fails, its counters get invalid, and the next responsible peer initializes its counter using the greatest timestamp of the committed patches at replica holders. Thus, the counter returns to its value before the update operation. Therefore, in the case of crash in this interval, continuous timestamping is assured.

- ***I_3*: after *I_2* and before sending the commit message to replica holders.** If the responsible peer fails in this interval, since the replica holders have not received the commit, they do not consider their received data as a valid replica. Thus, when the update requester checks the update, they answer that the update has not been done and the operation gets aborted. Therefore, in this case, continuous timestamping is not violated.

- ***I_4*: after *I_3* and before sending the terminate message to the update requester.** In this case, after detecting the failure or timeout, the update requester checks the status of the update in the DHT and finds out that the update has been done, thus it commits the operation. In this case, the update is done with a timestamp which is one unit greater than that of the previous update, thus the property of continuous timestamping is enforced.

6 Network Cost Analysis

In this section, we give a thorough analysis of CTRM's communication cost for both replica retrieval and update operations, and compare them with those of the same operations in two baseline services. Since usually the communicated messages are relatively small, we measure the communication cost in terms of the number of messages.

6.1 Replica Retrieval Cost

In section 3.2, we described the CTRM's operation for retrieving an up-to-date replica. In this section, we give a probabilistic analysis of this operation' cost in terms of the number of messages which should be communicated over the network.

The communication cost of replica retrieval in CTRM consists of the followings: 1) the cost of finding the group's responsible peer, denoted by c_g; 2) the cost of finding the first up-to-date replica at replica holders, denoted by c_{rh}; 3) the cost of returning the replica to the requester, denoted as c_{rt}. The first cost, i.e. c_g, consists of a lookup in the DHT which usually is done in $O(log\ n)$ messages where n is the number of peers of the DHT. For simplicity, we assume that the cost of a lookup is $log\ n$ messages.

The third cost, i.e. c_{rt}, takes simply one message because the first replica holder that maintains an up-to-date replica sends it directly to the requester.

The second cost, i.e. c_{rt}, depends on the number of replica holders which should be contacted for finding the first up-to-date replica. Let n_{rh} denotes the number of replica holders which should be contacted, then $c_{rt} = 2 \times n_{rh}$, i.e. for each replica holder we need one message to contact it and one message as answer. Thus, the total cost of retrieving an up-to-date replica by CTRM is $c_{ctrm} = (log\ n) + 2 \times n_{rh} + 1$. This cost depends on the number of peers in the system, i.e. n, and the number of replica holders which should be contacted by the group's responsible peer, i.e. n_{rh}.

Let us now give a probabilistic approximation of n_{rh}. Let p_{av} be the probability that a replica holder, which is contacted by the responsible peer, maintains an up-to-date replica. In other words, p_{av} is the ratio of the up-to-date replicas over the total number of replica holders, i.e. $|G_k|$. We give a formula for computing the expected value of the number of replica holders which should be contacted for finding the first up-to-date replica, in terms of p_{av} and $|G_k|$. Let X be a random variable which represents the number of replica holders which should be contacted. We have $Prob(X=i) = p_{av} \times (1-p_{av})^{i-1}$, *i.e.* the probability of having $X=i$ is equal to the probability that $i-1$ first replica holders do not maintain an up-to-date replica and the ith replica holder maintains an up-to-date one. The expected value of X is computed as follows:

$$E(X) = \sum_{i=0}^{|G_k|} i * \Pr ob(X = i)$$

$$E(X) = p_{av} * (\sum_{i=0}^{|G_k|} i * (1 - p_{av})^{i-1}) \quad (1)$$

Equation 1 expresses the expected value of the number of contacted replica holders in terms of p_{av} and $|G_k|$. Thus, we have the following upper bound for $E(X)$ which is solely in terms of p_{av}:

$$E(X) < p_{av} * (\sum_{i=0}^{\infty} i * (1 - p_{av})^{i-1}) \quad (2)$$

Because $p_{av} \leq 1$, by using the theory of series [3], we have the following equation:

$$\sum_{i=0}^{\infty} i * (1 - p_{av})^{i-1} = \frac{1}{(1-(1-p_{av}))^2} \quad (3)$$

Using Equations 3 and 2, we obtain:

$$E(X) < \frac{1}{p_{av}} \quad (4)$$

The above equation shows that *the expected value of the number of replica holders which should be contacted by the group's responsible peer is less than the inverse of the probability that a replica at a replica holder is up-to-date*.

Example. Assume that at retrieval time *50 %* of the replica holders have an up-to-date replica, *i.e.* $p_{av}=0.5$. Then the expected value for the number of replica holders to be contacted is less than 2, i.e. $n_{rh} \leq 2$. Thus, we have $c_{ctrm} \leq (log\ n) + 5$. In other words, in this example the total cost of replica retrieval in CTRM, i.e. c_{ctrm}, is close to the cost of doing one lookup in the DHT.

6.2 Data Update Cost

Let us now analyze the communication cost of the CTRM's for updating a data using the UCT protocol (described in Section 5.1) in terms of the number of messages. The communication cost consists of the followings: 1) the cost of finding the peer that is responsible for the group, denoted by c_g; 2) the cost of sending the patch to the replica holders, and receiving the acknowledges, denoted as c_{rh}; 4) the cost of committing or aborting the update operation, c_{cm}.

The first cost as shown in Section 6.1, is equal to doing a lookup in the DHT, thus can be estimated as $log\ n$ where n is the number of peers in the DHT. Let r be the number of replica holders, i.e. $r=|G_k|$. Then, the second cost, i.e. c_{rh}, is at most $r \times 2$. The third cost consists of sending a message to the replica holders and the requester, thus we have $c_{cm} = r + 1$. Thus, in total the communication cost of the update

operation by CTRM is $log\ n + 3 \times r + 1$ where r is the number of replicas and n the number of peers in the DHT.

6.3 Comparison with Baseline Services

Let us now compare the communication cost of CTRM with that of two baseline services. Although they cannot provide the same functionality as CTRM, the closest prior works to CTRM are the BRICKS project [13], denoted as BRK, and the Update Management Service (UMS) [1]. The assumptions made by these two works are close to ours, e.g. they do not assume the existence of powerful peers. BRK stores the data in the DHT using multiple keys, which are correlated to the data key. To find an up-to-date replica, BRK has to retrieve all replicas. UMS uses a set of m hash functions and replicates the data randomly at m different peers. To find an up-to-date replica, UMS has to make several lookups in the DHT.

Let r be the number of replicas and n the number of peers in the DHT. For updating a data, BRK and UMS perform r lookups in the DHT. Thus, the cost of update operation in these two services is $O(r \times log\ n)$. By comparing this cost with that of CTRM, i.e. $O((log\ n) + r)$ we see that the update operation in CTRM has a communication cost that is much lower than that of UMS and CTRM.

For retrieving an up-to-date replica, BRK has to retrieve all replicas. The cost of data retrieval in BRK is $O(r \times log\ n)$. The cost of data retrieval in UMS is $O(n_{cu} \times log\ n)$ where c_{cu} is the number of replicas which should be retrieved by using hash functions in order to find an up-to-date replica by UMS. In general, the value of n_{cu} is similar to the value of n_{rh} in CTRM. As we showed previously the communication cost of data retrieval by CTRM is $O(log\ n + n_{rh})$ which is lower than those of both BRK and UMS.

7 Experimental Validation

In this section, we evaluate the performance of CTRM through experimentation over a 64-node cluster and simulation. The experimentation over the cluster was useful to validate our algorithm and calibrate our simulator. The simulation allows us to study scale up to high numbers of peers (up to 10,000 peers).

7.1 Experimental and Simulation Setup

Our experimentation is based on an implementation of the Chord [21] protocol. We tested our algorithms over a cluster of 64 nodes connected by a 1-Gbps network. Each node has two Intel Xeon 2.4 GHz processors, and runs the Linux operating system. To study the scalability of CTRM far beyond 64 peers, we also implemented a simulator using SimJava. After calibration of the simulator, we obtained simulation results similar to the implementation results up to 64 peers.

Our default settings for different experimental parameters are as follows. The latency between any two peers is a random number with normal distribution and a mean of 100 ms. The bandwidth between peers is also a random number with normal distribution and a mean of 56 Kbps (as in [1]). The simulator allows us to perform tests with up to 10,000 peers, after which simulation data no longer fit in RAM and makes our tests difficult. Therefore, the default number of peers is set to 10,000.

In our experiments, we consider a dynamic P2P system, *i.e.* there are peers that leave or join the system. Peer departures are timed by a random Poisson process (as in [18]). The average rate, *i.e.* λ, for events of the Poisson process is λ=1/second. At each event, we select a peer to depart uniformly at random. Each time a peer goes away, another joins, thus keeping the total number of peers constant (as in [18]).

We also consider peer failures. Let *fail rate* be a parameter that denotes the percentage of peers that leave the system due to a fail. When a peer departure event occurs, our simulator should decide on the type of this departure, i.e. normal leave or fail. For this, it generates a random number which is uniformly distributed in [0..100]; if the number is greater than *fail rate* then the peer departure is considered as a normal leave, else as a fail. In our tests, the default setting for *fail rate* is 5% (as in [1]). In our tests, unless otherwise specified, the number of replicas of each data is 10.

In our tests, we compared our CTRM service with the replication management services in the BRICKS project [13], denoted as BRK, and the Update Management Service (UMS) [1]. Although they cannot provide the same functionality as CTRM (see Section 8), they are closest prior works to CTRM since their assumptions about the P2P system are similar to ours (as explained in Section 6.3).

7.2 Update Cost

Let us first investigate the performance of CTRM's update protocol. We measure the performance of data update in terms of response time and communication cost. By update response time, we mean the time needed to send the patch of an update operation to the peers that maintain the replicas. By update communication cost, we mean the number of messages needed to update a data.

Using our simulator, we ran experiments to study how the response time increases with the addition of peers. Using the simulator, Figure 4 depicts the total number of messages while increasing the number of peers up to 10,000, with the other simulation parameters set as defaults described in Section 7.1. In all three services, the communication cost increases logarithmically with the number of peers. However, the communication cost of CTRM is much better than that of UMS and BRK. The reason is that UMS and BRK perform multiple lookups in the DHT, but CTRM does only one lookup, i.e. only for finding the responsible peer. Notice that each lookup needs $O(\log n)$ messages where n is the number of peers of the DHT.

Figure 5 shows the update response time with the addition of peers up to 10,000, with the other parameters set as described in Section 7.1. The response time of CTRM is a little bit higher than that of UMS and BRK. The reason is that for guaranteeing continuous timestamping, the update protocol of CTRM performs two round-tips between the responsible peer and the other members of the group. But UMS and BRK only send the update actions to the replica holders by looking up the replica holders in parallel (note that the impact of parallel lookups on response time is very slight, but they have a high impact on communication cost). However, the difference in the response time of CTRM and that of UMS and BRK is small because the round-trips in the group are less time consuming than lookups. This slight increase in response time of CTRM's update operation is the price to pay for guaranteeing continuous timestamping.

7.3 Data Retrieval Response Time

We now investigate the data retrieval response time of CTRM. By data retrieval response time, we mean the time to return an up-to-date replica to the user.

Figure 6 shows the response time of CTRM, UMS and BRK with the addition of peers up to 10,000, with the other parameters set as defaults described in Section 7.1. The response time of CTRM is much better than that of UMS and BRK. This difference in response time can be explained as follows. Both CTRM and UMS services contact some replica holders, say *r*, in order to find an up-to-date replica, e.g. *r=6*. For contacting these replica holders, CTRM performs only one lookup (to find the group's responsible peer) and some low-cost communications in the group. But, UMS performs exactly *r* lookups in the DHT. BRK retrieves all replicas of data from the DHT (to determine the latest version), and for each replica it performs one lookup. Thus the number of lookups done by BRK is equal to the total number of data replicas, i.e. 10 in our experiments.

Let us now study the effect of the number of replicas of each data, say *m*, on performance of data retrieval. Figure 7 shows the response time of data retrieval for the three solutions while varying the number of replicas up to 30. The number of replicas has almost a linear impact on the response time of BRK, because to retrieve an up-to-date replica it has to retrieve all replicas by doing one lookup for each replica. But it has a slight impact on CTRM, because for finding an up-to-date replica CTRM performs only one lookup, and some low cost communications, i.e. in the group.

7.4 Effect of Peer Failures on Timestamp Continuity

Let us now study the effect of peer failures on the continuity of timestamps used for data updates. This study is done only for CTRM and UMS that work based on timestamping. In our experiments we measure *timestamp continuity rate* by which we mean the percentage of the updates whose timestamps are only one unit higher than

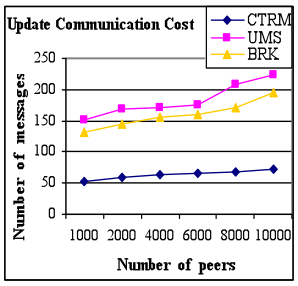

Figure 4. Communication cost of updates vs. number of peers

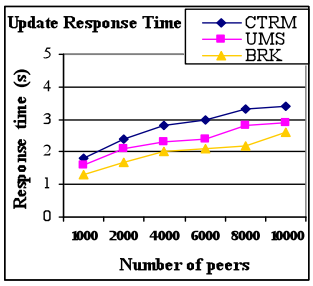

Figure 5. Response time of update operation vs. number of peers

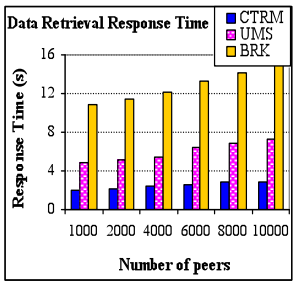

Figure 6. Response time of data retrievals vs. number of peers

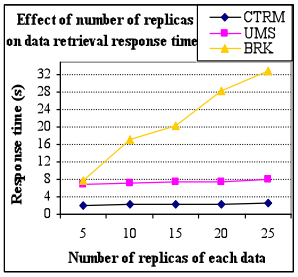

Figure 7. Effect of the number of replicas on response time of data retrievals

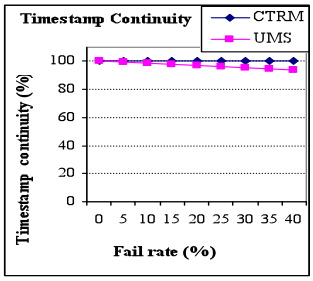

Figure 8. Timestamp continuity vs. fail rate

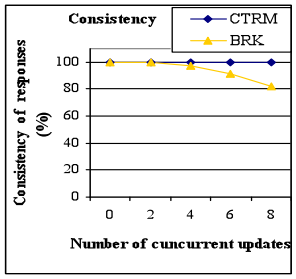

Figure 9. Consistency of returned results vs. number of concurrent updates

that of their precedent update. We varied the fail rate parameter, and observed its effect on timestamp continuity rate.

Figure 8 shows timestamp continuity rate for CTRM and UMS while increasing the fail rate, with the other parameters set as described in Section 7.1. The peer failures do not have any negative impact on the continuity of timestamps generated by CTRM, because our protocol assures timestamp continuity. However, when increasing the fail rate in UMS, the percentage of updates whose timestamps are not continuous increases.

7.5 Effect of Concurrent Updates on Result Consistency

In this section, we investigate the effect of concurrent updates on the consistency of the results returned by CTRM. In our experiments, we perform u updates done concurrently by u different peers using the CTRM service, and after finishing the concurrent updates, we invoke the service's data retrieval operation from n randomly chosen peers (n=50 in our experiments). If there is any difference between the data returned to the n peers, we consider the result as inconsistent. We repeat each experiment several times, and report the percentage of the experiments where the results are consistent. We perform the same experiments using the BRK service.

Figure 9 shows the results with the number of concurrent updates, i.e. u, increasing up to 8, and with the other parameters set as defaults described in Section 7.1. As shown, in 100% of experiments the results returned by CTRM are consistent. This shows that our update protocol works correctly even in the presence of concurrent

updates. However, the BRK service cannot guarantee the consistency of results in the case of concurrent updates, because two different updates may have the same version at different replica holders.

8 Related Work

In the context of distributed systems, data replication has been widely studied to improve both performance and availability. Many solutions have been proposed in the context of distributed database systems for managing replica consistency [4][15], in particular, using eager or lazy (multi-master) replication techniques, e.g. [12][14][24]. However, these techniques either do not scale up to large numbers of peers or raise open problems, such as replica reconciliation, to deal with the open and dynamic nature of P2P systems.

Most existing P2P systems support data replication, but without consistency guarantees. For instance, Gnutella [9] and KaZaA [11], two of the most popular P2P file sharing systems allow files to be replicated. However, a file update is not propagated to the other replicas. As a result, multiple inconsistent replicas under the same identifier (filename) may co-exist and it depends on the peer that a user contacts whether a current replica is accessed. In Freenet [6], the query answers are replicated along the path between the peers owning the data and the query originator. In the case of an update (which can only be done by the data's owner), it is routed to the peers having a replica. However, there is no guarantee that all those peers receive the update, in particular those that are absent at update time.

PGrid is a structured P2P system that deals with data replication and update based on a gossiping algorithm [8]. It provides a fully decentralized update scheme, which offers probabilistic guarantees. However, replicas may get inconsistent, *e.g.* as a result of concurrent updates, and it is up to the users to cope with the problem.

OceanStore [19] is a data management system designed to provide a highly available storage utility on top of P2P systems. It allows concurrent updates on replicated data, and relies on reconciliation to assure data consistency. The reconciliation is done by a set of high performance nodes, using a consensus algorithm. These nodes agree on which operations to apply, and in what order. However, in the applications that we address, the presence of such nodes is not guaranteed.

In [10], the authors present a performance evaluation of different strategies for placing the data replicas in DHTs. Our solution can be classified among those that use the neighbor replication strategy, i.e. that tries to place the replicas over the neighbors of a peer. However, our update protocol is new and not covered by any of the strategies described in [10].

The BRICKS project [13] provides high data availability in DHTs through replication. For replicating a data, BRICKS stores the data in the DHT using multiple keys, which are correlated to the data key, e.g. k. There is a function that, given k, determines its correlated keys. To be able to retrieve an up-to-date replica, BRICKS uses versioning. Each replica has a version number which is increased after each update. However, because of concurrent updates, it may happen that two different replicas have the same version number, thus making it impossible to decide which one is the latest replica.

In [1], an update management service, called UMS, was proposed to support data currency in DHTs, *i.e.* the ability to return an up-to-date replica. However, UMS does not guarantee continuous timestamping which is a main requirement for collaborative applications which need to reconcile replica updates. UMS uses a set of m hash functions and replicates randomly the data at m different peers, and this is more

expensive than the groups which we use in CTRM, particularly in terms of communication cost. A prototype based on UMS was demonstrated in [22].

9 Conclusion

In this paper, we addressed the problem of efficient replication management in DHTs. We proposed a new service, called continuous timestamp based replication management (CTRM), which deals with efficient data replication, retrieval and update in DHTS, by taking advantage of replica holder groups which are managed dynamically. We dealt with the dynamic behaviour of the group members, which can leave the system at any time. To perform updates on replicas, we proposed a new protocol that stamps the updates with timestamps that are generated using the replica holder groups. The updates' timestamps are not only monotonically increasing but also continuous. We analyzed the communication cost of CTRM, and show that its cost is very low in comparison to two baseline services in DHTs.

We evaluated CTRM through experimentation and simulation; the results show its effectiveness for data replication in DHTs. The results of our evaluation show that with a low overhead in update response time, CTRM supports fault-tolerant data replication using continuous timestamps. In our experiments, we compared CTRM with two baseline services, and the results show that data retrieval with CTRM is much more efficient than the baseline services. We investigated the effect of peer failures on the correctness of CTRM and the results show that it works correctly even in the presence of peer failures.

References

[1] Akbarinia, R., Pacitti, E., Valduriez, P.: Data Currency in Replicated DHTs. *ACM Int. Conf. on Management of Data (SIGMOD)*, 211-222, 2007.

[2] Akbarinia, R., Tlili, M., Pacitti, E., Valduriez, P., Lima, A.A.B.: Continuous Timestamping for Efficient Replication Management in DHTs. *Int. Conf. on Data Management in Grid and P2P Systems (Globe)*, LNCS Volume 6265, 38–49, 2010.

[3] Bromwich, T.J.I.: *An Introduction to the Theory of Infinite Series*. 3rd edition, Chelsea Pub. Co., 1991.

[4] Cecchet, E., Candea, G., Ailamaki, A.: Middleware-based database replication: the gaps between theory and practice. *ACM Int. Conf. on Management of Data (SIGMOD)*, 739-752, 2008.

[5] Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Shenker, S., Hellerstein, J.M.: A case study in building layered DHT applications. *ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 97-108, 2005.

[6] Clarke, I., Miller, S.G., Hong, T.W., Sandberg, O., Wiley, B.: Protecting Free Expression Online with Freenet. *IEEE Internet Computing 6(1)*, 40-49, 2002.

[7] Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-Area Cooperative Storage with CFS. *ACM Symp. on Operating Systems Principles*, 202-215, 2001.

[8] Datta, A., Hauswirth, M., Aberer, K.: Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. *IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 76-87, 2003.

[9] Gnutella. http://www.gnutelliums.com/.

[10] Ktari, S., Zoubert, M., Hecker, A., Labiod, H.: Performance evaluation of replication strategies in DHTs under churn. *Int. Conf. on Mobile and Ubiquitous Multimedia (MUM)*, 90-97, 2007.

[11] Kazaa. http://www.kazaa.com/.

[12] Krikellas, K., Elnikety, S., Vagena, Z., Hodson, O.: Strongly consistent replication for a bargain. *IEEE Int. Conf. on Data Engineering (ICDE)*, 52-63, 2010.

[13] Knezevic, P., Wombacher, A., Risse, T.: Enabling High Data Availability in a DHT. *Proc. of Int. Workshop on Grid and P2P Computing Impacts on Large Scale Heterogeneous Distributed Database Systems*, 363-367, 2005.

[14] Lin, Y., Kemme, B., Jiménez-Peris, R., Patiño-Martínez, M., Armendáriz-Iñigo, J.E.: Snapshot isolation and integrity constraints in replicated databases. *ACM Transactions on Database Systems (TODS)*, 34(2), 2009.

[15] Özsu, T., Valduriez, P.: *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1999.

[16] PalChaudhuri, S., Saha, A.K., Johnson, D.B.: Adaptive Clock Synchronization in Sensor Networks. *Int. Symp. on Information Processing in Sensor Networks*, 340-348, 2004.

[17] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. *ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 161-172, 2001.

[18] Rhea, S.C., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. *USENIX Annual Technical Conf.*, 127-140, 2004.

[19] Rhea, S.C., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: the OceanStore Prototype. *USENIX Conf. on File and Storage Technologies*, 1-14, 2003.

[20] Rowstron, A. I.T., and Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*, 329-350, 2001.

[21] Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F. Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. *ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 149-160, 2001.

[22] Tlili, M., Dedzoe, W.K., Pacitti, E., Valduriez, P., Akbarinia, R., Molli, P., Canals, G., Laurière, S.: P2P logging and timestamping for reconciliation. *PVLDB 1(2)*: 1420-1423, 2008.

[23] Xwiki Concerto Project: http://concerto.xwiki.com

[24] Wong, L., Arora, N.S., Gao, L., Hoang, T., Wu, J.: Oracle Streams: A High Performance Implementation for Near Real Time Asynchronous Replication. *IEEE Int. Conf. on Data Engineering (ICDE)*, 1363-1374, 2009.